Optimizing Memory Management in KULL with UMPIRE

Christopher Scarborough[1,2], Riyaz Haque[1]

[1]Lawrence Livermore National Laboratory, [2]California Polytechnic State University at San Luis Obispo

Abstract

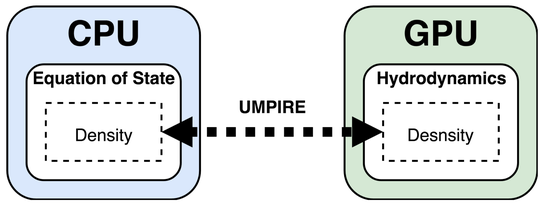
KULL is a complex multiphysics code that heavily uses advanced C++ language features including templates and virtual inheritance. It has historically been run only on homogeneous systems, but is transitioning to support the heterogeneous model of Sierra, a new supercomputer based on IBM Power 9 and NVIDIA Volta processors.

Complex data dependencies between physics packages complicate memory management in heterogeneous environments. We examine how UMPIRE, a portable API for memory management, can be used to manipulate memory in a platform-agnostic fashion and improve allocation performance through pooling with minimal overhead.

We demonstrate a 1.2% speedup in runtime using UMPIRE memory pooling on host-only systems, and do not find significant overhead from the library itself. The preliminary results show promise for significant gains when optimizing expensive device allocations.

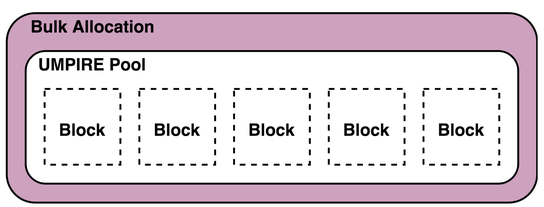
Pursuing Performance Portability

The transition to a host-device model has highlighted difficulties in maintaining high performance across machine architectures. KULL has many physics packages with different data locality needs.



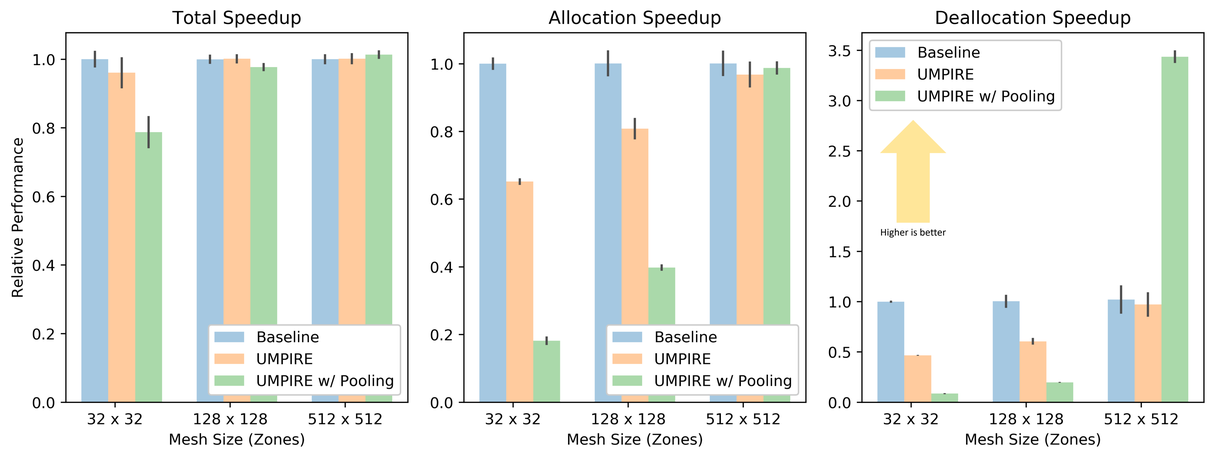
The Equation of State package might update a density field from the CPU, while the hydrodynamics package may need to access the same field from the GPU. Manually managing transfers is costly due to the many possible system configurations. UMPIRE abstracts these challenges by defining a platform-independent API to handle both allocations and movement.

Memory Pooling



UMPIRE allows users to define custom memory allocators, enabling straightforward implementations of memory pools. Pooling reduces fragmentation and overhead costs by pre-allocating large segments of fixed-size blocks that can be cheaply reassigned as needed.

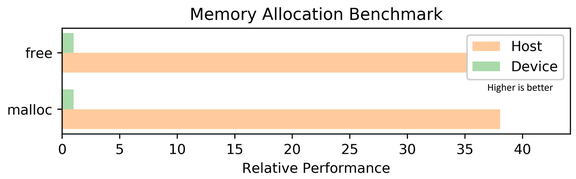
Performance Gained Through Pooling



Memory pooling offers a 1.2% speedup in host-only runtime over the baseline implementation for sufficiently large mesh sizes. However, it degrades performance by up to 25% on smaller meshes.

Profiling calls indicate the gains on large meshes are from decreased deallocation costs, while the losses on small meshes are distributed across allocation, deallocation, initialization, and destruction. One potential cause is unnecessary attempts to merge blocks of memory.

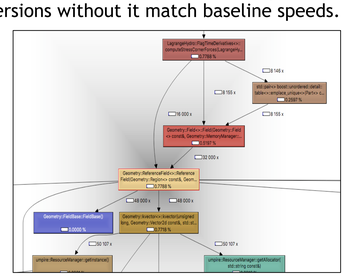
The data was averaged from 20 benchmarking runs on nodes in the `genie` cluster. Benchmarks used a `weekly-release` build of KULL with debugging symbols enabled. Runs were terminated after 100 timesteps. Longer runs of 1000 timesteps produced similar results.



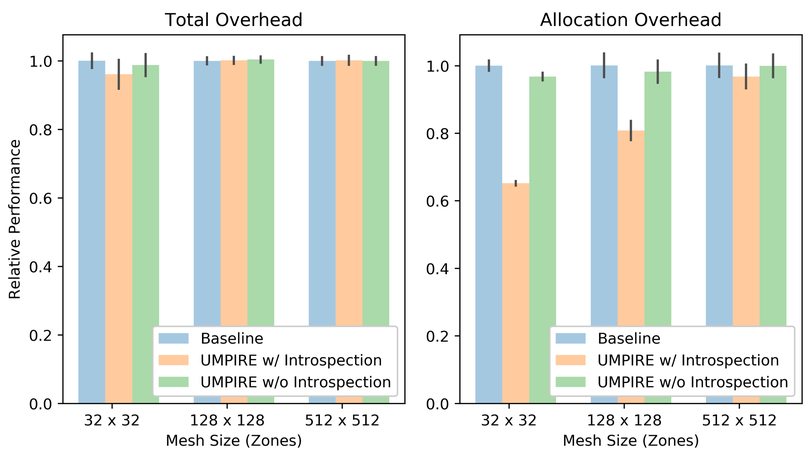
We expect to demonstrate a larger speedup when executing kernels on the graphics accelerator. Allocating host memory is over 35 times faster than device memory, so any reduction in the number of CUDA system calls required would have a substantial impact on runtime.

Overhead Introduced by Umpire

A second goal was to eliminate inefficiencies within in the UMPIRE library. Benchmarking indicates its pointer introspection capability incurs a substantial performance penalty, as versions without it match baseline speeds.



Analyzing UMPIRE's call graph with Valgrind determined that the UMPIRE version without introspection still incurred a 0.7% runtime penalty. 80% of the remaining overhead was caused by map lookups for allocator names.



The optimized build of UMPIRE disables pointer introspection and patches the search algorithm for free memory blocks to achieve comparable performance to the non-UMPIRE build in overall runtime and allocation speed. Much of the remaining delta stems from calls to `getAllocator`.

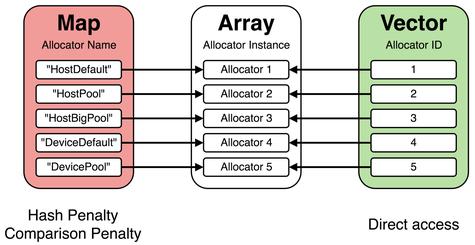
Conclusions

Memory pooling provides a small benefit on the CPU, at the cost of being very sensitive to parameters such as block size. However, it shows potential for applications with GPU accelerators, where memory operations are an order of magnitude more expensive.

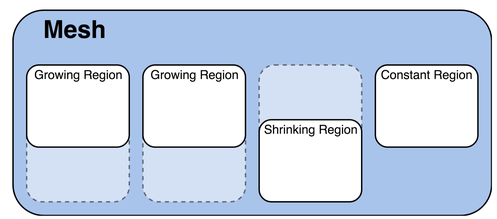
When optimized, UMPIRE introduces a small (0.7%) amount of overhead in KULL, which can be reduced further through tuning of its internal data management algorithms.

Future Work

UMPIRE currently relies on a map of allocator name strings to look up allocators. We expect that using an integer ID to index into a vector of allocators will improve performance and bring UMPIRE to within 0.2% of the reference malloc and free performance when executing on the host.



UMPIRE currently models data transactions between two execution spaces, the host and device. We wish to explore structural changes that model transactions as occurring between allocators, providing the flexibility to support both alternative strategies for memory management and novel machine architectures, such as utilizing an NVME cache on the host.



KULL supports the concepts of regions, or processing areas smaller than the full mesh. It is not feasible to have every region stored at the size of the full mesh due to memory limitations, so they may increase or decrease in size as the simulation progresses. We wish to explore heuristics for determining when and how to reallocate memory for region-based fields.

References

1. J.A. Rathkopf et al. *KULL: LLNL's ASCI Inertial Confinement Fusion Simulation Code.* Pittsburgh, PA: American Nuclear Society, 2000.
2. https://wci.llnl.gov/simulation/computer-codes
3. https://github.com/LLNL/Umpire

IM-POST-755978

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52 07NA27344